AFIT/EN-TR-91-7

Air Force Institute of Technology

Migrating a C-based CAD Tool to an Object-Oriented Database/C++ Environment: Conversion Costs and Performance Analysis

Mark A. Roth, Maj, USAF
Timothy M. Jacobs, Capt, USAF

6 December 1991

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

Migrating a C-based CAD Tool to an Object-Oriented Database/C++ Environment: Conversion Costs and Performance Analysis[†]

Mark A. Roth and Timothy M. Jacobs

Department of Electrical and Computer Engineering (AFIT/ENG)
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433

December 6, 1991

Abstract

This paper examines the potential of object-oriented databases to support complex design applications. To do this we migrated the Magic very large scale integrated (VLSI) circuit design tool written in the C language to a new environment in which Magic's existing file management code is replaced with a C++ language interface to the commercial object-oriented database management system (OODBMS) product ObjectStore. In our initial implementation we found the performance of this tool as implemented on the OODBMS to be marginally faster than the tool as currently implemented with a flat file system in several critical areas. Increased functionality, including version management, multi-user concurrency control, and recovery, are now possible with the converted system. However, we found the conversion process itself time consuming and fraught with software engineering perils; the final product is not significantly more or less maintainable. We conclude that the conversion of large, complex systems should not be undertaken without experienced programmers nor without a pressing need for increased database functionality. Conversion of such systems to improve performance alone should be avoided.

1 Introduction

Database management systems (DBMS) have proven themselves in a large variety of computer applications. Today's commercial DBMSs provide an effective tool for managing large repositories of data while providing access to multiple users and applications. All users have access to the same data since it is managed in a single environment. Concurrency control and recovery methods ensure data consistency despite multiple users and hardware or software failures. Applications can be easily added without knowledge of the physical layout of the data. Overall, modern DBMSs provide numerous advantages over alternate data management facilities.

Despite these many advantages, there are numerous computer applications which continue to spurn the use of a DBMS in favor of their own unique application file system. Among these are

[†]This research was supported by a grant from Rome Laboratories, RL/OCTS, Griffis AFB, NY

engineering design systems which are heavily dependent on large amounts of computer data. Few applications systems which support these design processes are integrated with a DBMS.

Although conventional databases are unable to adequately support these applications, the need for some sort of database support becomes evident as the systems proliferate among more powerful workstations and in increasingly complex engineering environments. Object-oriented database management systems (OODBMS), while still not widely available, have shown the potential for providing the necessary database support for these complex, data intensive applications.

This study examines the potential of object-oriented databases to support complex design applications. To do this we migrated the Magic very large scale integrated (VLSI) circuit design tool [9] written in the C language to a new environment in which Magic's existing file management code is replaced with a C++ language interface to the commercial OODBMS product ObjectStore [11]. We compared the performance of this tool as implemented on the OODBMS to its performance as currently implemented with a flat file system, and draw conclusions about whether an OODBMS can adequately support a complex, data intensive, automated design system.

In our initial implementation we found the performance of this tool as implemented on the OODBMS to be marginally faster than the tool as currently implemented with a flat file system. Interactive use of the tool (loading cells, expanding cell hierachies, creating cells and cell instances) generally ran more than 50 percent faster. These operations ran in fractions of a second in both systems, so the improvement was not that noticeable. Other operations such as design rule checking and program initialization were slower with the OODBMS, but these operations took a long time with the original system as well. The new system retains nearly all of the same functionality as the original system but now new functionality, including version management, multi-user concurrency control, and recovery, are now possible. We found the conversion process itself time consuming and fraught with software engineering perils; the final product is not significantly more or less maintainable. We conclude that the conversion of large, complex systems should not be undertaken without experienced programmer analysts nor without the need for increased database functionality.

2 Objectives

The primary objective of this study was to determine whether or not an object-oriented database management system provides the performance and functionality necessary to support a typical computer-aided design tool. To fulfill this objective, the design tool must maintain all functionality provided by the existing file management system. Performance of the design tool must remain acceptable to the tool users. This is specified as no more than a ten percent increase in response time over the current implementation.

In this study we also wished to demonstrate that conversion of a design tool application from a unique flat file management system to an OODBMS is not a difficult or time intensive task. To meet this goal, the time and effort spent converting to the OODBMS must be cost effective with respect to the increased utility of a database management system and the reduction in future software maintenance costs.

3 OODBMS Performance Benchmarks

Much of the literature on database performance evaluation addresses the results of standard benchmarks as applied to various DBMSs. While most of these benchmarks reflect typical applications for relational DBMSs, Cattel has developed an approach for measuring the performance of object-oriented database systems [3]. Before discussing his approach, however, Cattel points out that "The most accurate measure of performance for engineering applications would be to run an actual application, representing the data in the manner best suited to each potential DBMS [3:364]."

Cattel proposes a database of parts on a circuit board with connections between them. He summarizes the three most important measures of performance in an object-oriented DBMS as:

Lookup and Retrieval. Look up and retrieve an object given its identifier.

Traversal. Find all objects referenced by or connected to a selected object.

Insert. Insert objects and their relationships to other objects.

To meet the performance requirements of engineering applications, Cattel suggests that a DBMS must be able to perform 1000 random operations per second. He noted that none of the OODBMS implementations in research or production environments met his criteria when his paper was written in 1987.

Berre and Anderson's *HyperModel* benchmark [2] presents a similar approach to performance measurement. In addition to the operations proposed by Cattel, the HyperModel benchmark includes:

Sequential Scan. Visit each object in the database sequentially.

Closure Operations. Perform operations on all objects reachable by a certain relationship from a specified object.

Open-and-Close. Time to open and close the database.

4 The ObjectStore Database Management System

Because of the complexity of object-oriented databases and the immaturity of the field, few commercially available object-oriented databases exist. Of those that do exist, many are merely object-oriented interfaces to a relational database. Only recently have any databases been commercially released with memory management techniques suitable for object-oriented database management. One such database management system is ObjectStore, an object-oriented database management system released in 1990, with an upgrade release in 1991, by Object Design, Incorporated.

ObjectStore supports most of the object-oriented database characteristics listed in [1]. The database design language, *C++*, provides support for complex state, inheritance, and user defined data types [15]. Specific views for an object can be expressed in the *C++* functions associated with that object. ObjectStore also provides a versioning mechanism to support long transactions and multiple versions. To handle large amounts of data, ObjectStore uses a memory mapping and page swapping mechanism which can be customized by the database designer [11]. ObjectStore does

not support schema evolution so any change to a schema makes data created with the old schema unreadable by the new program [14].

In addition to its uniquely object-oriented characteristics, ObjectStore also has traditional database management facilities. All access to the database must occur within a transaction. The concurrency control protocol uses two-phase locking. Data integrity is ensured through relations and inverse relations which synchronize related objects when one of them is modified. ObjectStore provides tools for managing collections (groups of homogeneous objects) and supports query processing over these collections.

ObjectStore's *virtual memory mapping architecture* is key to its performance. This architecture allows persistent data stored in ObjectStore to be handled in the same way as non-persistent (transient) data. Large amounts of data can be retrieved and manipulated with minimal overhead through virtual memory management. When referenced data is not in main memory, a page fault occurs which is intercepted by ObjectStore so that it can retrieve the data from the database into memory. The overall effect of the memory mapping architecture is to provide the developer a single view of memory—basically expanding the program memory to the size of the database.

For an application to work with ObjectStore three auxiliary processes are required—the *ObjectStore Server*, the *Directory Manager*, and the *Cache Manager*. The Server handles all storage and retrieval of persistent data. The Directory Manager manages ObjectStore directories much as Unix manages its directories. The Cache Manager controls swapping of data between the cache memory associated with an application and the virtual database memory.

ObjectStore provides interfaces to both *C* and *C++*. It also has its own Data Definition and Manipulation Language (DML) which is a superset of *C++*. The DML simplifies *C++* library routines, such as setting the database root and controlling transactions, by replacing a sequence of *C++* commands with a single DML command.

5 The Magic VLSI Layout Design System

Due to the complexity and cost associated with the design and creation of VLSI circuits, computer-assisted tools are essential. One of the key steps in the circuit development process is the design of a physical layout of the circuit which can be directly implemented on a chip. Tools for manipulating and verifying this design are necessary to keep track of the numerous components and connections and to minimize the risk of the chip failing after it has been manufactured. One such tool is the Magic VLSI layout system which was originally developed at the University of California in Berkeley with the latest release from Digital Equipment Corporation's Western Research Laboratory in 1990.

The purpose of Magic is to increase the power and flexibility of previous layout editors so that designs can be entered quickly and modified easily [13]. To accomplish this goal, Magic provides basic commands for creating, copying, modifying, and deleting circuit components along with capabilities for automated circuit routing and continuous checking of design rules. Circuits can be created completely from scratch or through hierarchical inclusion of any number of sub-components. File extraction tools have also been included as part of Magic for compatibility with circuit testing and manufacturing systems. [9]

To improve performance and simplify the designer's view of a circuit, Magic implements some unique features. The geometrical contents of a circuit are represented using a technique called *corner stitching*. In this technique, a circuit contains a number of corner-stitched planes, each of which consists of a number of rectangular tiles representing the physical material to be included

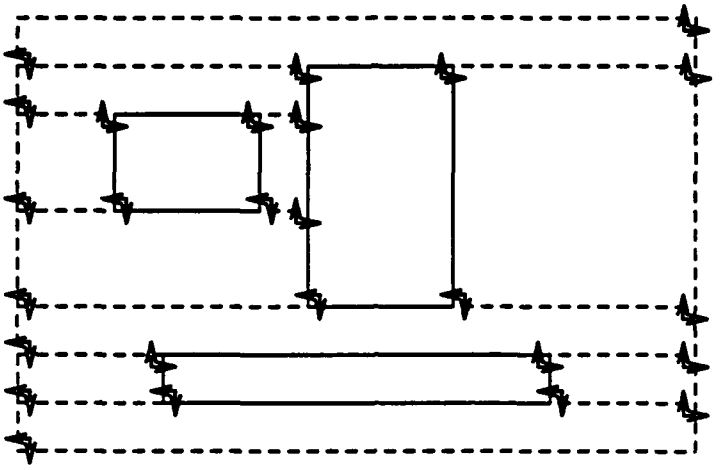

Figure 1: Corner stitching of tiles in a plane [13]

in the actual circuit. These tiles are the fundamental data units represented in the database. Each tile is linked in its lower-left corner to those tiles to its left and bottom. Another link in the upper-right corner connects the tiles to the right and top. Figure 1 demonstrates how three filled-in tiles (enclosed with solid lines) would be stitched together with blank tiles (dashed lines) in a single plane. The corner stitches provide a form of two-dimentional sorting, permiting search operations to be performed more efficiently [13]; however, iterating through all tiles in a cell may require traversing a number of subcell hierarchies in the database.

6 Converting Magic for ObjectStore

6.1 Design Recovery of Magic

Magic is a large software system consisting of over 250,000 lines of *C* source code in over 40 separate Unix directories. To maximize code efficiency, many of the data structures and algorithms are extremely complex, often using obscure *C* language characteristics. The design documentation consists of a maintainer's manual with a brief description of the directory layout and functionality along with in-line comments in the source code.

In approaching a design recovery of the Magic software system, the first step was to review the existing documentation. This review revealed a combination of functional and object-oriented problem decomposition. The object-oriented modules, such as the *window manager* and *database manager*, encompass all data structures and services for an object completely within the module. Other modules like *plot, plow*, and *wiring* include all procedures necessary for accomplishing the designated function. Of greatest importance to this study were the database management module and its associated functions. Most interfaces to the ObjectStore database management system will take place within this module. Furthermore, while most modules interface with the database manager, implementation with ObjectStore affects only a few of these interfaces.

Analysis of the *C* header files for the database manager revealed the primary data structure shown in Figure 2. Here an object is represented by a box with the object name in the top section of

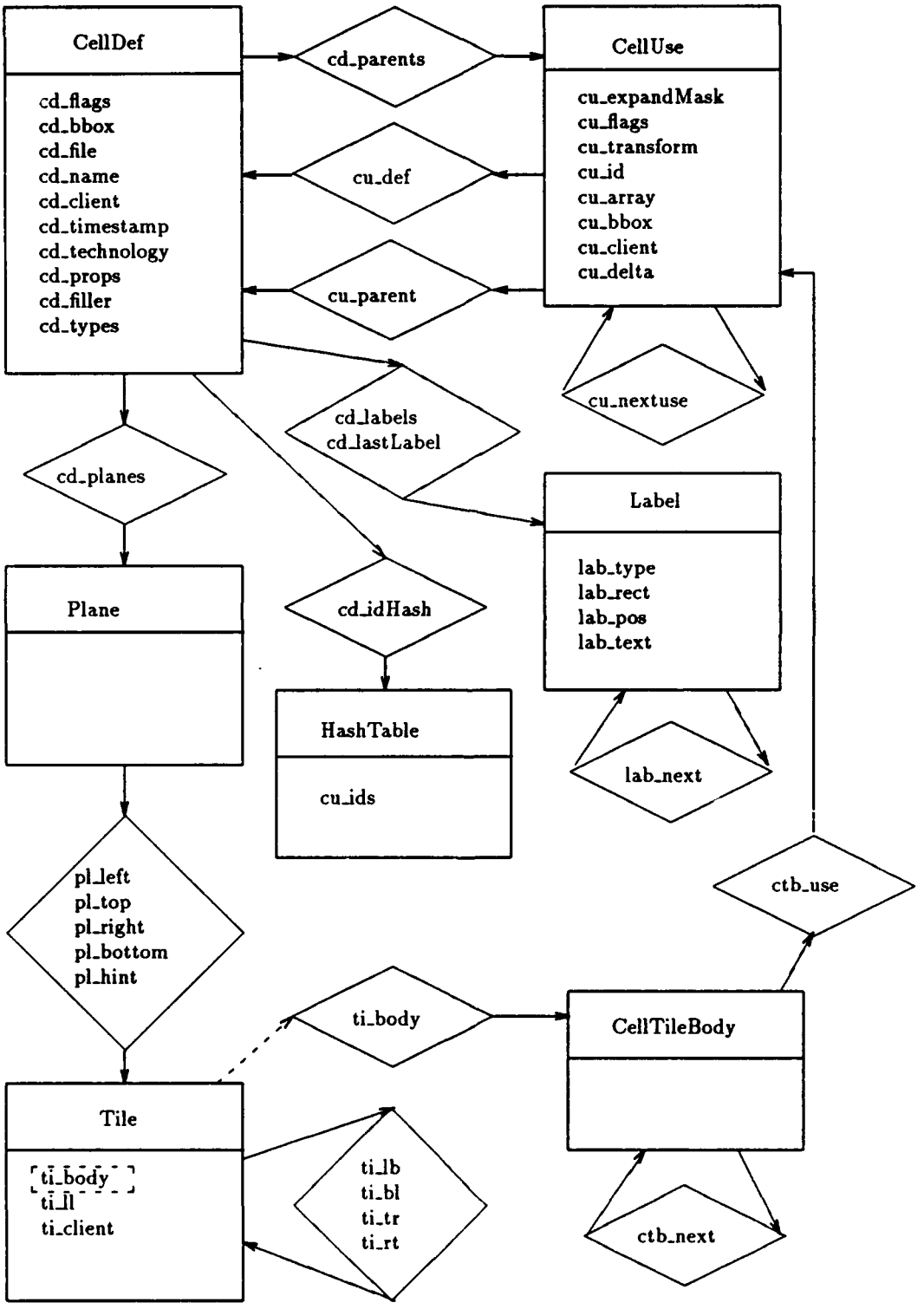

Figure 2: Data structure for Magic cells

the box and its attributes in the lower section. Diamonds represent relationships between objects. The key component of a Magic circuit is the *cell definition* (`CellDef`). This includes descriptive fields such as the cell name, associated chip technology, and time of last update. It also includes pointers to the *planes* which contain the geometrical representations in the cell; a pointer to a list of *labels* associated with the cell; a pointer to a list of cell instances (referred to in Magic as *cell uses*); and a pointer to a hash table of all instances of other cells which reference the cell. Each plane also has pointers to a corner-stitched list of *tiles* which are contained in the plane. Normally `ti_body` specifies the paint in the tile; however, tiles in the subcell plane include a list of pointers to the subcell uses which overlap the tile (`CellTileBody`). All cell definitions currently active in Magic are contained in `dbCellDefTable`, a hash table which is visible only to the database manager.

To better understand the relationship between cell definitions and cell uses, Figure 3 provides a simplified example. Cells `A` and `B` each represent cells that have cells `X` and `Y` as subcells. The cell definition of `X` is named `CdX`. It references cell use `CuX3` with its `cd_parents` pointer. It also points to a hash table which only includes cell use `CuX1`. This indicates that `CdX` is not referenced by any cells other than itself. Cell use `CuX3` is the cell use associated with cell `B`. The `cu_parent` pointer to cell definition `CdB` shows this relationship. The `cu_def` pointer in `CuX3` points to cell definition `CdX`, of which `CuX3` is a specific instance. Each cell use also has a `cu_nextuse` pointer which points to the next cell use associated with a particular cell definition. In this case, `CuX3` points to `CuX2` which points to `CuX1` where the list terminates with a null pointer. `CuX2` represents the instance of `CdX` in cell `A` and `CuX1` represents the instance of `CdX` associated with cell `X`. Each cell definition always has an instance associated with itself. Cell definition `CdA` provides a better example of the `cd_idHash` pointer. In `CdA` this points to a hash table of all cell uses that are included in cell `A`. This includes an instance of the cell itself (`CuA1`) as well as each of its subcells (`CuX1` and `CuY1`).

After additional frequent use of the Unix *grep* and *calls* commands, we determined the functions of the database module most likely to be affected by implementation with ObjectStore.

- Create and delete cell definitions and cell uses
- Create and maintain the cell definition table
- Write and read cells to and from disk
- Create, split, join, and delete cell planes and tiles

If the database module was truly object-oriented, design recovery would end here. Unfortunately, many modules throughout Magic directly access the data structures of the database module, thereby making the interface less well defined. For instance, the initialization routines in many modules directly access planes within a cell. Similarly, each window has a cell instance associated with it which the window manager modifies without going through the database module.

6.2 Restructuring of Magic Software to Work with ObjectStore

In our initial work, the only modifications made to the Magic software were those necessary for Magic to work with the ObjectStore database management system. This required persistently allocating all transient database structures and providing an entry point into the ObjectStore database. In addition, the ObjectStore database file must be opened and closed and transactions must be specified such that all database accesses occur within a transaction. This first phase of

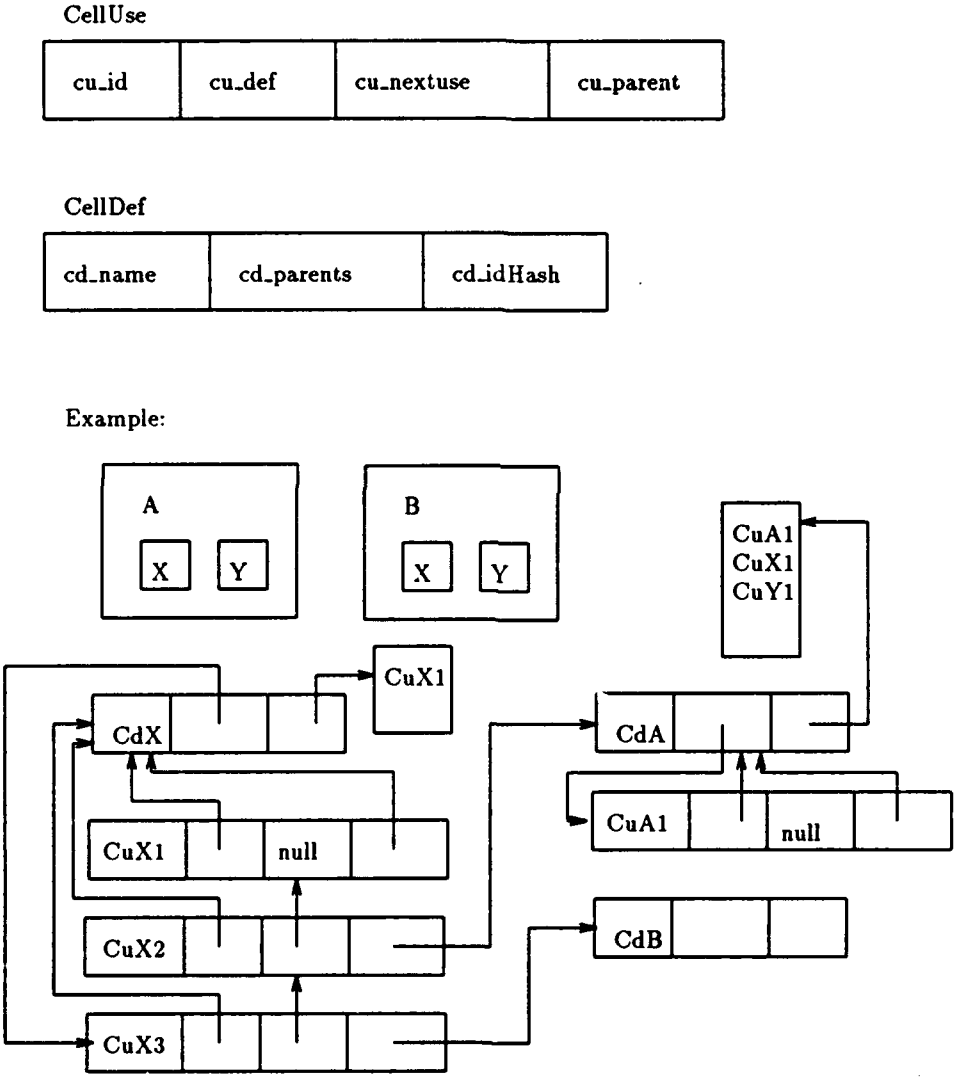

Figure 3: Relationship between cell definitions and cell uses

implementing Magic with ObjectStore required the following changes. (A complete summary of all modifications to the Magic software is contained in [6]).

- Where a database element such as those in Figure 2 has been allocated with the procedure **MALLOC**, remove the **MALLOC** and replace it with the ObjectStore Data Definition and Manipulation Language (DML) persistent **new** command. The **new** command specifies in which database and in which configuration the element should be stored.

```
/* replace MALLOC with ObjectStore DML new
 *
 * MALLOC(CellDef *, cellDef, sizeof (CellDef));
 */

cellDef = new(magicdb1, cellConfig) CellDef;
```

  Similarly, where a database element is deallocated with **FREE**, remove the **FREE** command and replace it with a DML persistent **delete** command.

```
/* replace old memory deallocation with DML delete
 *
 * FREE (cellUse->cu_id);
 */

delete cellUse->cu_id;
```

- Make the cell definition symbol table (**dbCellDefTable**) the entry point into the database. This is accomplished by declaring it as a **persistent** variable.

```
persistent<magicdb1> osHashTable *dbCellDefTable = NULL;
```

- Declare and initialize (open) the database in the program **main.c** and include the main procedure within an ObjectStore transaction. The original attempt at accomplishing this was to enclose the contents of *magicMain* (the main Magic procedure) within a DML **do_transaction** statement; however the main Magic procedure is terminated from within another module, thus preventing the entire main procedure from executing. As a result, the **do_transaction** statement never ends and no data is written to the database. Resolution required separate transactions to initialize Magic and another transaction (**main_tx**) for the main Magic process. Note that a different format is used for the main transaction. ObjectStore allows a transaction to be enclosed within the **do_transaction** statement or started with **transaction::begin** and ended with **transaction::commit** (to save the results of the transaction) or **transaction::abort** (to restore the database to its state prior to the transaction).

```
/* Open database "/osmagic/magicdb1" */
magicdb1 = database::open("/osmagic/magicdb1",0,0664);

do_transaction() {
    workspace::set_current(user_ws);
}  /* end of transaction */

  /* begin initialization transaction */
do_transaction() {

    mainInitBeforeArgs(argc, argv);
    mainDoArgs(argc, argv);
    mainInitAfterArgs();
}  /* end of initialization transaction */

  /* begin main transaction called "main_tx" */
main_tx = transaction::begin(transaction::update);


TxDispatch( (FILE *) NULL);

mainFinished();
```

- Close the database and commit the main *transaction* in the procedure *MainExit.*

```
transaction::commit(main_tx);
magicdb1->close();
```

In an object-oriented *C++* program, implementation of these changes might have been rather straightforward and uncomplicated. Unfortunately, Magic was not such a program. The first difficulty was that Magic is written in *C* rather than *C++*. *C++* was designed to be compatible with *C* [15]; however, this compatibility is not complete. ObjectStore has a *C* library interface, but this does not allow one to take full advantage of object-oriented programming techniques. Any program which contains ObjectStore DML must be compiled with the DML compiler. This compiler is a slightly modified *C++* compiler. Thus all procedures in a program containing ObjectStore DML, whether affected by ObjectStore or not, must be modified for compatibility with *C++*. This requires type specification of all function parameters. In many cases, Magic passes function pointers as parameters, which complicates type specification. Also, for a *C++* procedure to be linked with a *C* program, the linkage must be specifically defined with an **extern** "C" type specification for the function declaration. Thus, all of the forty plus header files containing *C++* function declarations had to be modified to include the **extern** "C" qualifier. This was accomplished by defining a preprocessor constant of `_OSMAGIC` in a header file. Other header files are then modified to include the **extern** "C" qualifier for DML programs if `_OSMAGIC` is defined or just the qualifier **extern** if `_OSMAGIC` is not defined. Figure 4 shows modified code from `tile.h` which demonstrates the changes required for each header file.

We also had to modify other procedures outside of the database module. Additional storage is allocated for a cell in the string duplication program of the utilities module. A persistent version of this program was required for any string duplication within the cell structure. The cell labeling, cell painting, and cell subroutine programs also allocate cell storage which must be persistent.

```
#ifndef _OSMAGIC
/* if using old magic code, use "C" function declarations */
/* Jacobs 02/08/91 */

extern Plane *TiNewPlane();
extern void TiFreePlane();
extern Tile *TiSplitX();
extern Tile *TiSplitY();

#else
/* use function declarations modified for compatibility with C++ */
/* Jacobs 02/08/91 */

extern "C" Plane *TiNewPlane(Tile*, CellConfig*);
extern "C" void TiFreePlane(Plane*);
extern "C" Tile *TiSplitX(Tile*, int);
extern "C" Tile *TiSplitY(Tile*, int);

#endif
```

Figure 4: Sample header file modifications from **tile.h**

Since the cell symbol table is used as a database entry point, it must also be persistently allocated. The hash table abstract data type (ADT), of which the cell symbol table is an instance, is also used for non-database functions, so a separate, persistent hash table ADT had to be created and used for the cell symbol table. This symbol table only uses strings as keys, so the *C* **union** in the original hash table ADT can be replaced with a string. This simplifies DML implementation by eliminating the need for union discriminant functions.

The maze router (**mzrouter**) initialization procedure attempts to directly access planes within a cell. The first time the initialization procedure is run on the database, the cell and planes become persistently allocated. The **mzrouter** initialization procedure must then be modified to access the persistent planes by traversing the cell hierarchy.

Modifications up to this point have been necessary for Magic to work with persistently allocated structures in an ObjectStore database. Previously these structures were transiently allocated and could be cleared by quitting Magic or by reading the cell again. Persistent allocation eliminates the possibility of deleting a cell or removing changes that are unwanted by simply quitting or re-reading. To provide these functions and other input/output functions in a manner similar to the original Magic system, ObjectStore's versioning capabilities are required.

Versioning requires definition of a **configuration** to be versioned and persistent allocation of **workspaces** to control the versions. For this implementation of Magic, the global workspace contains the latest frozen version of a cell, much as the flat disk file does in the original version of Magic. New cells are created and modified in the user workspace. The user workspace is set as the current workspace for the duration of the Magic program.

The obvious choice for a version configuration is a cell definition and all of its subcomponents and subcells. A symbol table must be created for the cell configurations so that each cell definition can be associated with its configuration. Whenever a cell is read, it is checked out of the global

workspace into the current user workspace. Writing a cell requires checking the cell back into the global workspace. To flush a cell, the version in the current user workspace is destroyed and the old version is checked out of the global workspace to replace the destroyed version in the user workspace. New procedures which search the configuration symbol table and check the appropriate configuration in or out of the current workspace are needed.

Even with versioning, some commands could not be made to work exactly like the original Magic system. Both the *write* and *save* command write a cell to disk. The *save* command allows the cell to be saved under another name. Both of these commands allow further modification of the cell after writing. These commands must be modified to work with ObjectStore. *Save* now checks the cell back into the global workspace, but allows continued editing of the cell by checking it back out. *Write* checks the cell into the global workspace and prohibits further editing.

In the original Magic, cells are deleted by using the Unix *rm* command external to Magic. This is not possible using ObjectStore so the new command *remove cellname* had to be added. This command only deletes a cell definition if it is not used by any other cells.

6.3 Code Instrumentation for Performance Measurement

To compare the performance of Magic with and without ObjectStore, a method had to be provided for measuring this performance. The *Sun* operating system provides profiling options which are implemented by the compiler. This profiling provides detailed timing and usage statistics on every function in Magic. Unfortunately, due to the size of Magic, interpretation of this information is time intensive and complicated. Since such detailed information was not necessary, the code itself was instrumented at critical points to measure only that information which is essential for comparing performance of the two versions of Magic.

All commands and button input are processed through the command interpreter module. The `TxDispatch` function dispatches all Magic text and button commands. This allows timing statistics to be initiated prior to calling the command processor or button handler and terminated immediately following the call. Similarly, initialization statistics were measured by surrounding Magic's initialization procedures with statistical commands. A separate procedure, written for gathering statistics, made calls to the Unix functions *getrusage* and *gettimeofday* and calculated processing time and wall clock time. This information was printed to a file along with the command being processed or the button handler in use.

6.4 Testing

To test functionality of the new system, we utilized the Magic tutorial which walks a new user through all of the basic commands that are available. In addition, since performance testing concentrates on the database these tests serve to further validate the functionality.

Performance testing compares the differences in access time between the Magic implementation using ObjectStore and the original implementation using flat Unix files. This testing must measure Magic performance during a typical user session along with concentrated testing of the database functions of Magic. A typical user session (in our academic setting) does not require many database accesses. This is likely due to the lack of common database functions (e.g., no method exists in Magic for searching for an existing cell among the flat files) and the experimental nature of academic research which leads to circuits built entirely from scratch. Since a typical user session does not adequately test the database capabilities, performance comparisons were also conducted

using the HyperModel Benchmark guidelines (see Section 3). The HyperModel Benchmark lists six areas which are important for measuring database performance—lookup and retrieval, traversal, insertion, sequential scan, closure operation, and opening and closing of the database.

All performance testing was accomplished with existing Magic commands. As such, some of the six areas above were only partially tested. Use of existing commands was necessary so ObjectStore performance could be directly compared to the existing flat file structure. The ObjectStore database is loaded in advance with all Magic cells in a specific search path, so that the search space is roughly equivalent with that of the Unix directories.

Performance testing was accomplished on two existing cells. One cell contains 87 subcells with eight levels of nesting and the other cell contains two subcells with one level of nesting. The various Magic commands for measuring the benchmark criteria are discussed below.

- Look up and retrieve an object from the database. The *load cellname* command searches the database until it finds the specified cell and then displays it in the selected window. If the cell has any subcells, these are not initially displayed.

- Traverse pointer hierarchy. The *expand* command loads and displays all of the subcells in a selected area of the root cell. When the entire cell is selected, all of the subcell pointers are dereferenced and the subcells displayed.

- Insert an item into the database. The *getcell cellname* command loads a subcell from the database and creates a new instance of that cell in the root cell. The *load* command without a specified cell creates a new cell definition in the database. With the non-persistent version of the database, the new cell definition or instance is not actually created until the cell is saved; therefore, the time to write modified cells to disk must be included in the comparison with the ObjectStore version.

- Closure operations. In normal operation, the Magic design rule checker runs in the background. To test closure, however, the design rule checker is turned off, a subcell is added on top of the existing cell, and the design rule checker is run on the entire cell.

- Sequential Scan. Since no use is made of ObjectStore collections, this aspect of the database benchmark is not tested.

- Database initialization. Since the existing Magic system has no database, this benchmark can not be directly compared; however, the tests are still run and compared based on the overall time to initialize Magic.

7 Results

The primary objective of this study was to show that an object-oriented database can provide the performance and functionality necessary to support a computer-aided design tool. By careful application of ObjectStore utilities, complete functionality of the Magic VLSI circuit layout system has nearly been obtained. In this section we compare the relative performance of Magic as implemented with ObjectStore to the original flat file data management system. We also point out the difficulties encountered while converting the complex *C* code of Magic to work with the complicated, object-oriented data management facilities of ObjectStore.

7.1 Performance

To compare the performance of Magic using ObjectStore to Magic's performance using its original flat file system, the performance tests described in Section 6.4 were accomplished on a Sun Sparcstation II with 32MB of memory using two different Magic databases:

drfm This database consists of 110 objects (i.e., cell definitions) and 1861 different instances (i.e., cell uses) of these objects. There are over 78,000 fundamental data items (i.e., tiles) at the lowest level of object nesting. This database was tested using cell **drfmchip** with 87 subcells and eight levels of nesting. The Magic display of this cell, with all subcells completely expanded, appears in Figure 5.

tutorial This database contains 70 objects and 103 different instances. There are nearly 9300 fundamental data items at the lowest level of object nesting. Testing was accomplished using cell **tut4a** which contains two subcells and one level of nesting. Figure 6 shows the Magic display of the tiles and subcell structure of this cell.

The results of the performance tests are shown in Tables 1 and 2. Raw performance test results are contained in [5]. The results shown in Tables 1 and 2 represent averages of all valid results obtained for each command. For more accurate comparison with ObjectStore, the resources necessary for writing all modified cells to disk are added to the performance results for the insertion and closure tests.

For the **drfmchip** cell, instance insertion and closure are tested on a subcell nested four levels deep in the hierarchy (**big_nandmux**) and on a subcell nested eight levels deep (**mcell10**). There are 16 instances of the **big_nandmux** subcell and 6144 instances of the **mcell10** subcell. Between these two levels of nesting, the time difference to accomplish instance insertion and closure was considerable (a factor of 20). The average of these two different levels is used for comparison in Table 1. Instance insertion and closure are tested on a subcell nested only one level deep in the **tut4a** cell. There are only eight instances of this subcell.

Testing of the **drfm** database was accomplished with ObjectStore cache sizes of both 640 and 2048 sectors. The 640 sector cache worked more quickly for look up and retrieval, traversal, initialization, and object insertion. These are the results shown in Table 1. Similarly, the results for the 2048 sector cache are used for instance insertion and closure. All ObjectStore results for the **tutorial** database were obtained with a cache size of 2048 sectors. Neither size of cache performed better for all test cases. Optimal cache size depends on which database utilities are more commonly used.

The commands used for performance testing fall into three different categories. Those used for *look up and retrieval* and *hierarchy traversal* require reading from the database. In the original version of Magic, *insert* and *closure* commands are performed entirely within memory. For this reason, the resources necessary to write to disk all cells modified by these commands are added to the results in Tables 1 and 2. *Initialization* is a combination of reading and writing from the database and initializing memory. In general, ObjectStore had better response time than Magic's original data management system for those commands with frequent database access relative to total processing time.

ObjectStore's performance varies most significantly with the *closure* command (i.e., *drc catchup*). To understand why, more insight into the design rule checker (*drc*) is required. The design rule

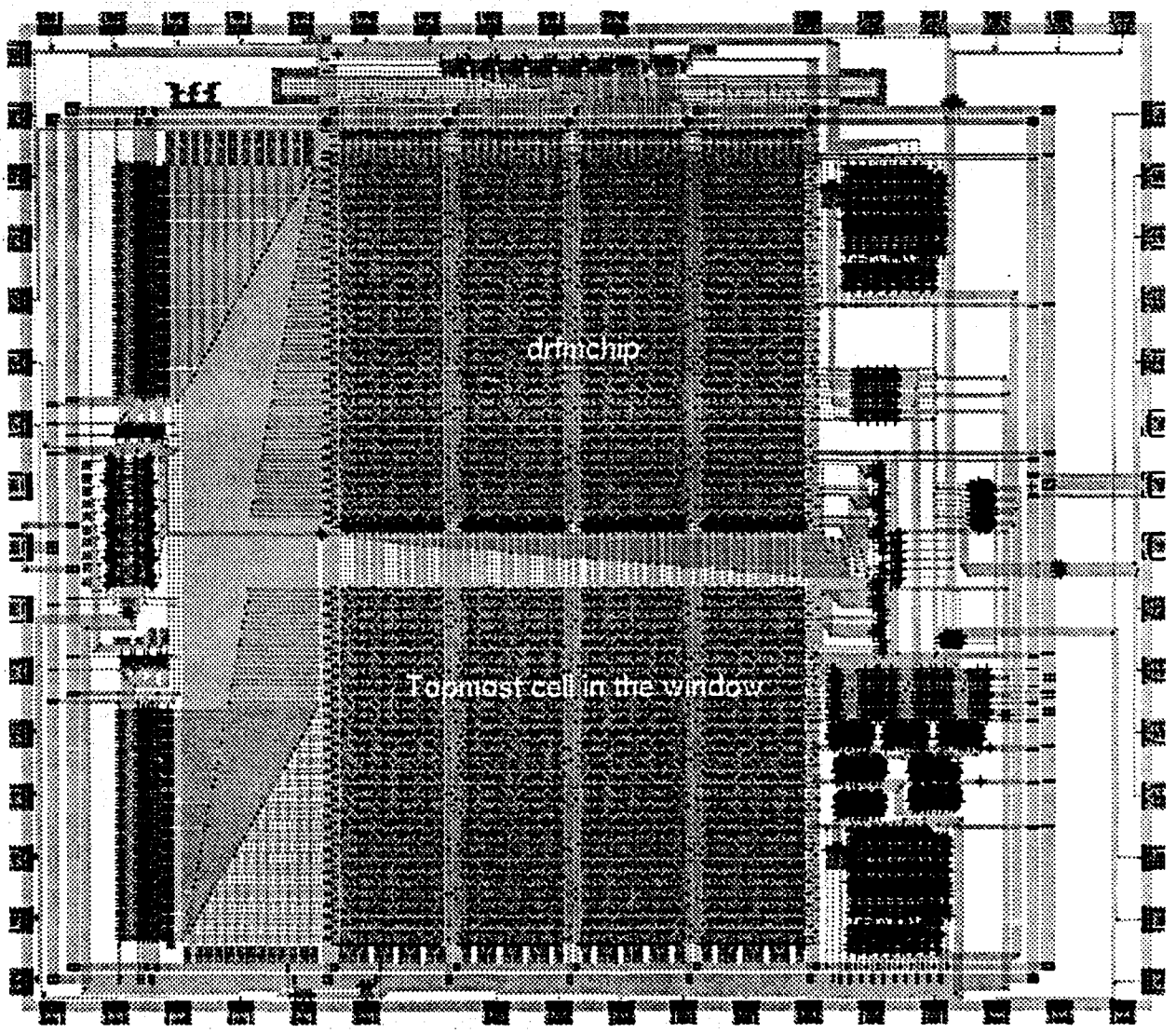

Figure 5: Magic display of cell **drfmchip**

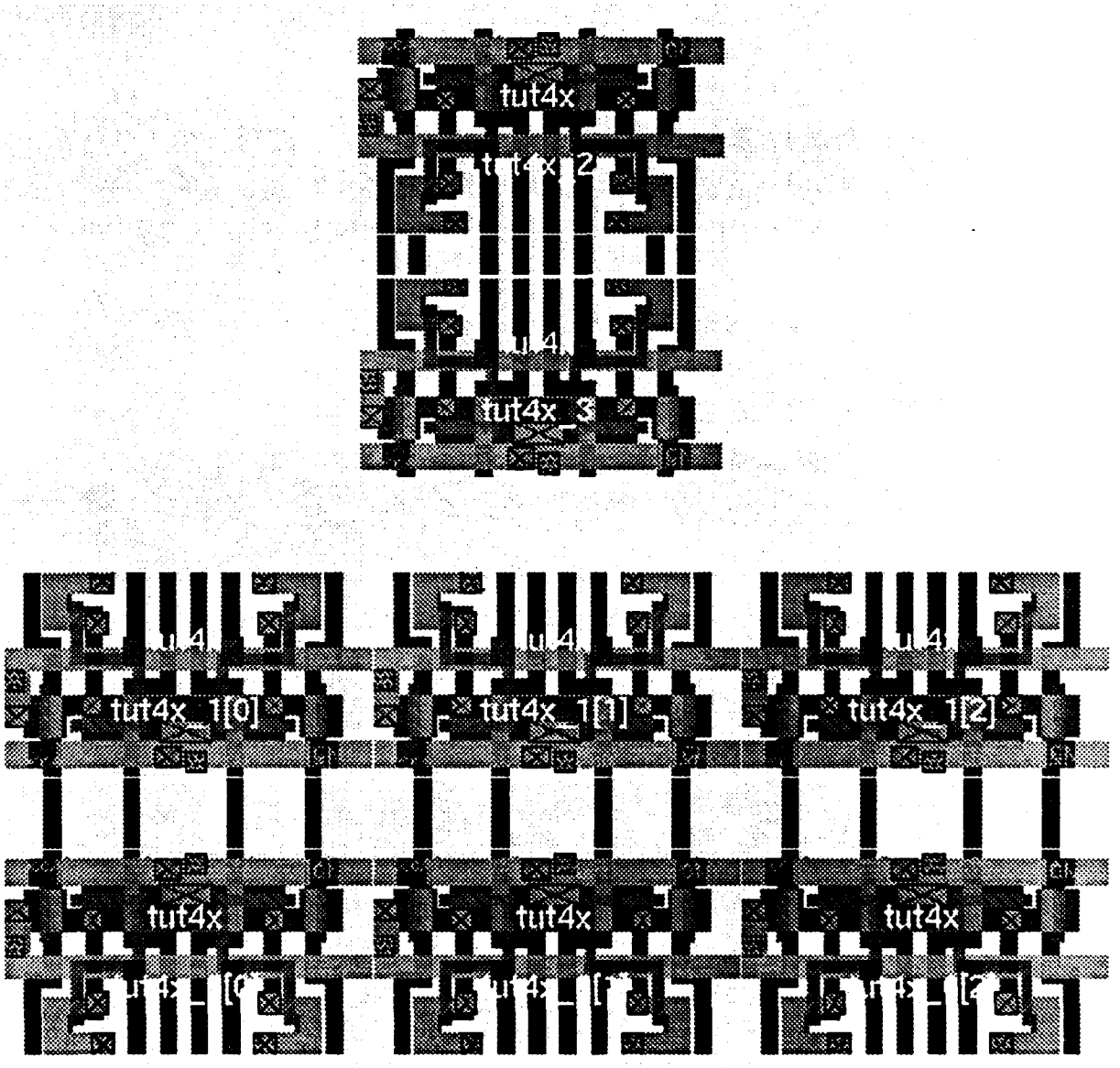

Figure 6: Magic display of cell **tut4a**

Criteria Tested *Command Used*	Resource Measured	Data Management System Flat File	 ObjectStore	Percent Change
Look up/retrieve	CPU time (seconds)	0.06	0.05	-17
load drfmchip	Elapsed time (seconds)	0.12	0.33	+175
Hierarchy Traversal	CPU time (seconds)	6.62	1.54	-76
expand	Elapsed time (seconds)	9.32	4.37	-53
Insert (object)	CPU time (seconds)	0.03	0.02	-33
load test	Elapsed time (seconds)	0.124	0.040	-68
Insert (instance)	CPU time (seconds)	1.36	0.22	-84
getcell test	Elapsed time (seconds)	2.63	0.24	-91
Closure	CPU time (seconds)	52.19	193.42	+271
drc catchup	Elapsed time (seconds)	53.98	197.66	+266
Initialization	CPU time (seconds)	5.74	5.86	+2
	Elapsed time (seconds)	10.71	15.21	+42

Table 1: Benchmark performance results for **drfm** database

Criteria Tested *Command Used*	Resource Measured	Data Management System Flat File	 ObjectStore	Percent Change
Look up and retrieve	CPU time (seconds)	0.01	0.03	+200
load tut4a	Elapsed time (seconds)	0.047	0.037	-21
Hierarchy Traversal	CPU time (seconds)	0.04	0.02	-50
expand	Elapsed time (seconds)	0.0788	0.0252	-68
Insert (object)	CPU time (seconds)	0.020	0.017	-15
load test	Elapsed time (seconds)	0.158	0.026	-84
Insert (instance)	CPU time (seconds)	0.060	0.013	-78
getcell test	Elapsed time (seconds)	0.532	0.029	-95
Closure	CPU time (seconds)	0.58	2.00	+245
drc catchup	Elapsed time (seconds)	1.05	2.03	+93
Initialization	CPU time (seconds)	5.66	5.82	+3
	Elapsed time (seconds)	8.53	9.91	+16

Table 2: Benchmark performance results for **tutorial** database

Data Item	Size in Kbytes		Percent
	Flat File	ObjectStore	Change
`tutorial` database	58	713	+1129
`drfmchip` database	724	4882	+574
initialized database (empty)	0	147	$+\infty$
`test` object & instance	0.081	2.3	+2740

Table 3: Comparison of ObjectStore and Unix flat file disk usage

checker applies rules from a file of over 500 lines to each tile in the cell. Hierarchical designs are checked by ensuring the cell alone is consistent, and that the combination of the cell and all of its subcells is consistent [12]. This may require traversing the cell hierarchy a number of times to complete the design rule checking; thus, small discrepancies in response time are multiplied into large differences such as those shown in Tables 1 and 2.

The design rule checker usually runs in the background because of the large amount of processing it requires [12]. It limits its checks to those cells which are currently in memory; other cells are checked the next time they are read into memory. This allows incremental application of design rules to the cell and eliminates the need to process the entire cell at once. Since the ObjectStore version of Magic extends memory to include database items which are on the disk, it always appears as if the entire database is in memory; thus, the design rule checker checks all cells in the database. Because of these variations in virtual memory, our comparisons are made with the entire cell resident in main memory.

The ObjectStore database required considerably more disk space than Unix flat files (see Table 3). Both ObjectStore and Magic add overhead to the ObjectStore database which is not saved in the Unix file representation. For Magic this overhead is approximately 50 kilobytes. The ObjectStore overhead varies with the total size of the database. With no data other than the Magic and ObjectStore overhead in the database, the total database size is nearly 150 kilobytes. In addition, ObjectStore holds the entire data structure for each cell in memory. This includes pointers and empty hash table buckets. The Unix files only contain the contents of the cell data structure. Because of the large amount of overhead associated with ObjectStore, the difference is disk usage is more significant for smaller data items.

7.2 Conversion Experience

One of the objectives of this study was to show that conversion of a design system to an object-oriented database is a cost-effective venture. If this new technology is to replace existing design systems, system managers must be convinced that the benefits of using a database will outweigh the costs of conversion. While performance is one of the key issues in this decision, there are a number of benefits and costs associated with converting to a database management system.

7.2.1 Problems Encountered

Due to the size and complexity of Magic, and the intricacies of ObjectStore, a number of difficulties were encountered during the conversion process. Some of these difficulties were simply due to the immaturity of the ObjectStore database system and the lack of documentation for Magic.

Implementation of ObjectStore's versioning capability presented the only problem which could not be resolved. The `configuration::of(`*object*`)` command is designed to return a pointer to the configuration of the object specified. In a number of places this pointer is then used to allocate a subobject in the same configuration as the object. The sequence of command is as follows:

```
cellConfig = configuration::of(cellDef);
cellUse = new(magicdb1, cellConfig) CellUse;
```

ObjectStore fails when attempting to allocate the subobject. No fix[1] was currently available for this problem, so versioning was not tested in this study.

Another problem area stems from the incompatibility of *C* and *C++*. In theory, *C++* is designed to be a superset of *C* [15]. While this may be true in a *C* program that is very well designed and coded, in reality *C* allows many structures that are not compatible with *C++*. While Magic is a rather well designed system, its size and the complex programming structures which it uses to maximize efficiency have led to code which is unstructured and difficult to trace. Some common problems include functions used without being previously declared, data types used in header files that are declared in a separately included header file, and functions with different types of parameters being passed as parameters to another function. Since *C++* has much stricter type checking, these problems had to be resolved if the program included any ObjectStore DML commands which required the *C++* compiler.

While not directly related to compatibility between *C* and *C++*, the ability to cast types in *C* presented significant problems. Magic uses this capability extensively. In one particular instance, `ti_body` is declared as a `char`; however, to handle subcells within a plane, a list of pointers to `CellUse` is cast to this variable. Initially, Magic sets the `ti_body` field to a `char`. If ObjectStore encounters a `char` value when it is expecting a persistent pointer, it is unable to dereference the pointer and the program aborts. Such errors are very difficult to trace due to the use of type casting.

Many difficulties were encountered due to peculiarities of the ObjectStore DML. Those which were hardest to resolve are listed below.

- Inconsistency of schemas. If the database schema is modified, the old database is no longer valid for that application [14]. A procedure must be made for converting the old data to match the new schema, or the data is lost.[2]

- Incompatibility of persistent types with *C*. Persistent types have an extra level of indirection which must be accounted for if using them in a *C* program [14]. ObjectStore also uses a persistent dereferencing type which is not compatible with the *C* compiler. This requires all *C* programs to access persistent data types through a *C++* interface.

- Declarations for database pointers and persistent database entry points cannot be nested within a procedure or function. Since Magic is broken into a number of subdirectories, it was initially unclear at exactly what level these declarations should occur.

- When using versioning, the effects of `configuration::forget` are unclear. This command should remove the specified version from the current workspace [10]; however, this does not

[1] As this was being written, we received version 1.2 of ObjectStore which reportedly fixes this problem.

[2] The next major release of ObjectStore (version 2.0) will reportedly include some form of schema evolution capability.

	Estimated	Actual
Magic design recovery	10	11
ObjectStore Familiarity	10	11
C++ Conversion	0	13
ObjectStore Persistence	15	17
ObjectStore Versions	10	20
Total	45	72

Table 4: Man days spent converting Magic to work with ObjectStore

happen when the versioned object has just been created and exists in no other workspace. The `configuration::destroy_version` command was used instead. To be able to destroy a version, however, the configuration must have been previously frozen in some workspace. This is accomplished by first checking any new configuration into the global workspace before checking it back out to make modifications.

- A discriminant function is required when using a `union` [10]. No explanation is given on what this function should do and when it should be called. To eliminate the associated complexities, the `union` was removed as it was no longer necessary for the updated Magic program.

- When more data is read into memory than ObjectStore can manage, it attempts to evict a page from memory causing an exception which crashes Magic. To overcome this problem when testing `drfmchip`, the cache size had to be manually increased to 2048 sectors.

As with any product, improved documentation would have saved us a lot of time in dealing with the above problems. Suggestions for documentation writers include:

- In addition to telling us how to implement a function give us an explanation as to why it is done that way so that when the circumstance are different, the proper adaptation can be made.

- When multiple ways of performing a function are available, as with ObjectStore's library and DML interfaces, give adequate examples of both techniques and correlate the functions performed in each technique.

7.2.2 Effort Required

This study implements a minimal database version of Magic. Changes were made to replace the original flat file structure with an ObjectStore database while maintaining the same functionality. No attempt was made to take full advantage of ObjectStore's other features. The time spent on this conversion is broken down in Table 4. The estimates in this chart are based on the perceived difficulty of each process and the total time available for the conversion.

As reflected in this table, the conversion to ObjectStore took longer than expected. Since *C* constructs are supposed to be compatible with *C++*, the only code conversions expected were those necessary for the ObjectStore DML. The many other *C/C++* conversions which were actually required had a significant impact on the time required to make Magic work with ObjectStore and

its *C++* compiler. The other major deviation in time is that required for implementing ObjectStore versions. This is an extremely challenging ObjectStore utility and the documentation is somewhat limited.

8 Conclusion

The objectives of this study required that the ObjectStore version of Magic provide the full functionality of the original version. Response time must not increase by more than ten percent over the original version. This study also attempted to demonstrate that conversion of Magic from its flat file data management system to ObjectStore is a cost effective undertaking. The results of Section 7 are analyzed in the following subsections to determine whether these study objectives were met.

8.1 Database Functionality

The ObjectStore version of Magic retains nearly all of the same functionality as the original system. Minor changes in cell creation and deletion were necessary, but these changes actually improve the system since less reliance is placed on correct user manipulation of the Unix file system. Furthermore, new functionality, including version management, multi-user concurrency control, and recovery, are now possible. Instead of using version management to simulate quitting without saving or re-reading a previous cell over what was previously transient memory, versions can now become an integral part of the design process. Versioning, together with new multi-user capabilities, will allow Magic to be ported to a cooperative work environment with, say, multiple students working on complex design project simultaneously. In a similar way, recovery of previous versions, or of current work during a system crash, is now possible with true database management system support.

8.2 Database Performance

The results presented in Section 7 show Magic to meet performance goals for only three of the six areas of performance benchmark testing: hierarchy traversal and object and instance insertion. Because of this, one may tend to conclude that ObjectStore's performance is inadequate for supporting complex engineering design systems such as Magic. Cattel suggests, however, that benchmark performance tests may not be an accurate measure of performance; rather "The most accurate measure of performance for engineering applications would be to run an actual application ..." [3:364].

When actually using Magic, the difference in performance was not readily apparent. For look up and retrieval, the difference in response time, while representing an increase of nearly 200 percent, was still only measured in fractions of a second—barely perceptible to a human user. Database initialization, while taking 42 percent more time with ObjectStore, is only performed once per user session. Five seconds in a session that may be hours long does not seriously detract from overall performance. Closure operations, as tested with the *drc catchup* command, took considerably longer with ObjectStore. Even with the original version of Magic, however, this command took a long time to accomplish. It is for this reason that the designers of Magic expect users to generally run the design rule checker in the background. This background checking can be turned off when a

large number of changes are made to a circuit design. When the changes are completed, *drc catchup* will run the design rule checker on all changes made during the session. Again, this represents a few minutes of trade-off in performance during what typically is an hours long session.

ObjectStore required a considerable amount of space to store the Magic databases. In the original environment in which the performance tests were accomplished, this amount of space had a significant impact. With the current implementation of Magic using a single transaction, no segment of the database could be found to remove from memory once the entire circuit was swapped into memory. The single transaction implementation coupled with the large database size led to memory quickly being used up and the Magic system failing. These problems do not necessary reflect poorly on object-oriented databases, however, since proper transaction implementation would easily overcome the problems. In addition, newer releases of ObjectStore are projected to better handle such memory swapping problems [14].

When considering the performance of the ObjectStore version of Magic, one must realize that Magic was not designed using object-oriented programming techniques. Any optimization built into the Magic code is designed to improve efficiency of the Unix flat file storage system, not a database management system. The fact that the original Magic design does not take advantage of these fundamental concepts of the ObjectStore database management system may account for the failure of Magic to perform as well with ObjectStore as it does with its current Unix flat file management system. That the ObjectStore version of Magic performs best for hierarchy traversal demonstrates what Object Design considers to be the primary benefit of ObjectStore's architecture [8:61].

Overall, the performance results obtained from implementing Magic with ObjectStore are promising. Benchmark tests indicate that ObjectStore performance falls within the ten percent increase criteria for only three of six areas. Actual usage of Magic, however, showed that the areas not meeting the performance criteria are unlikely to noticeably impact the overall performance of Magic. Modification of Magic's design to better take advantage of ObjectStore's database features would likely improve performance. Similarly, memory limitations of the existing implementation provided for a very unstable environment; however, proper implementation of transactions along with projected upgrades to the ObjectStore database system would likely eliminate this problem.

8.3 Usefulness of Conversion vs. Cost

One of the primary costs of any software system is that necessary for software maintenance. These costs can be minimized if software changes affect only small, localized segments of code and if maintainers can easily understand the organization and functionality of existing code. Object-oriented computing attempts to minimize the impact of changes through data encapsulation, in which the underlying data is accessible only through a well-defined interface [16]. Increased understanding is attained through abstraction, a concept in which the programmer's model of an object more closely approximates the user's conceptual model of the object [4]. A database management system also supports data encapsulation and abstraction by providing mechanisms for defining storage structures and manipulating information [7].

ObjectStore supports both object-oriented computing and database management. With its persistent data structures and procedures for managing data, all input and output procedures can be eliminated from the program. This eliminates the complexity involved with transforming data from its flat file representation to the appropriate data structure in memory. Unfortunately,

the implementation of Magic for this study does not take full advantage of ObjectStore's object-oriented computing facilities. Because the interface to the database is not well-defined, a change in the database structure may affect many segments of the system. By not converting Magic's existing *C* code to *C++* which is used by ObjectStore, additional complexity was added since both a *C* and a *C++* interface must be specified for each module.

ObjectStore provides the capability to greatly enhance the maintainability of Magic. Unfortunately, by failure to take full advantage of ObjectStore's object-oriented facilities, and through offsetting the maintenance advantages of a database management system with an extra interface for *C++*, the overall maintainability of Magic remains about the same. No significant maintenance cost savings are realized with the version of Magic implemented for this study.

On the other hand, the costs associated with converting to ObjectStore were relatively high. In four months of intense study and programming with ObjectStore, it was not possible to learn and understand every aspect of ObjectStore or to even completely understand any single aspect. No programming at all was accomplished using ObjectStore's collection and relation facilities and we were not able to test the versioning features.

Another significant cost associated with conversion was modifying *C* programs for compatibility with *C++*. This task could have been avoided by using ObjectStore's *C* library interface; however, this would increase the effort required to take advantage of the object-oriented programming facilities of *C++* at a later time.

The ObjectStore implementation of Magic for this study was not cost effective. The costs associated with learning the ObjectStore system and making *C* programs compatible with *C++* were not offset by any significant improvement in maintainability. A complete redesign using object-oriented techniques which take advantage of ObjectStore's data definition and manipulation language (DML) would significantly increase the understandability of the Magic code and likely reduce future maintenance costs. Such a redesign would also likely improve Magic's performance. The costs of a complete redesign would be high, however, suggesting that ObjectStore may be better suited for developing new systems or converting systems that are already object-oriented rather than converting a complex system design such as Magic's.

Overall performance of Magic as implemented for this study does not yet justify conversion of existing, complex C-language applications to use object-oriented databases. Many questions are left unanswered due to the difficulties encountered while implementing Magic with ObjectStore. To answer these questions and take better advantage of ObjectStore's facilities, we plan on continuing an object-oriented conversion of Magic with ObjectStore in a continuing study. A better transaction model, use of ObjectStore collections and indices, and repaired version management may tip the scales toward conversion. It is quite possible that complete implementation of these proposals would lead to a Magic system with significantly improved functionality, maintainability, and performance. We, in fact, were quite impressed that the ObjectStore version of Magic matched performance with the flat file version in areas where significant user time is spent, even with our less than optimal initial conversion.

References

[1] ATKINSON, M., BANCILHON, F., DEWITT, D., DITTRICH, K., MAIER, D., AND ZDONIK, S. The object-oriented database system manifesto. In *First International Conference on*

Deductive and Object-Oriented Databases (Kyoto, Japan, 1989), W. Kim, J.-M. Nicolas, and S. Nishio, Eds., Elsevier.

[2] BERRE, A. J., AND ANDERSON, T. L. The hypermodel benchmark for evaluating object-oriented databases. In *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991, ch. 5, pp. 75–91.

[3] CATTEL, R. Object-oriented DBMS performance measurement. In *Proceedings of the 2nd Workshop on OODBS* (1988), pp. 364–367.

[4] HEILER, S., ET AL. An object-oriented approach to data management: Why design databases need it. In *24th Design Automation Conference Proceedings* (1987), pp. 335–340.

[5] JACOBS, T. M. An object-oriented database implementation of the Magic VLSI layout design system. Master's thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, Dec. 1991.

[6] JACOBS, T. M. *OSmagic Programmers' Manual.* Air Force Institute of Technology, Dec. 1991.

[7] KORTH, H. F., AND SILBERSCHATZ, A. *Database System Concepts.* McGraw-Hill Book Company, New York, 1986.

[8] LAMB, C., ET AL. The ObjectStore database system. *Communications of the ACM 34* (Oct. 1991), 50–63.

[9] MAYO, R. N., ET AL. *1990 DECWRL/Livermore Magic Release*, 1990.

[10] OBJECT DESIGN, INC. *ObjectStore Reference Manual.* Burlington, Massachusetts, Mar. 1991.

[11] OBJECT DESIGN, INC. *ObjectStore User Guide.* Burlington, Massachusetts, Mar. 1991.

[12] OUSTERHOUT, J. K. *Magic Tutorial #6: Design-Rule Checking.* University of California, Berkeley, CA, 1990.

[13] OUSTERHOUT, J. K., ET AL. Magic: A VLSI layout system. In *21st Design Automation Conference Proceedings* (1984), pp. 152–159.

[14] SAWYER, C., AND TURNER, S. Object Design, Inc. Technical Support, June - October 1991. Multiple telephone conversations.

[15] STROUSTRUP, B. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[16] ZDONIK, S. B., AND MAIER, D., Eds. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	6 December 1991	Technical Report

4. TITLE AND SUBTITLE
Migrating a C-based CAD Tool to an Object-Oriented Database/C++ Environment: Conversion Costs and Performance Analysis

5. FUNDING NUMBERS

6. AUTHOR(S)
Mark A. Roth, Maj, USAF
Timothy M. Jacobs, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/EN-TR-91-7

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
RL/OCTS
Rome Labs
Griffis AFB, NY 13441

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper examines the potential of object-oriented databases to support complex design applications. To do this we migrated the Magic very large scale integrated (VLSI) circuit design tool written in the C language to a new environment in which Magic's existing file management code is replaced with a C++ language interface to the commercial object-oriented database management system (OODBMS) product ObjectStore. In our initial implementation we found the performance of this tool as implemented on the OODBMS to be marginally faster than the tool as currently implemented with a flat file system in several critical areas. Increased functionality, including version management, multi-user concurrency control, and recovery, are now possible with the converted system. However, we found the conversion process itself time consuming and fraught with software engineering perils; the final product is not significantly more or less maintainable. We conclude that the conversion of large, complex systems should not be undertaken without experienced programmers nor without a pressing need for increased database functionality. Conversion of such systems to improve performance alone should be avoided.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Object-Oriented Databases, C, C++, Performance Measurement, Conversion Cost	26
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL